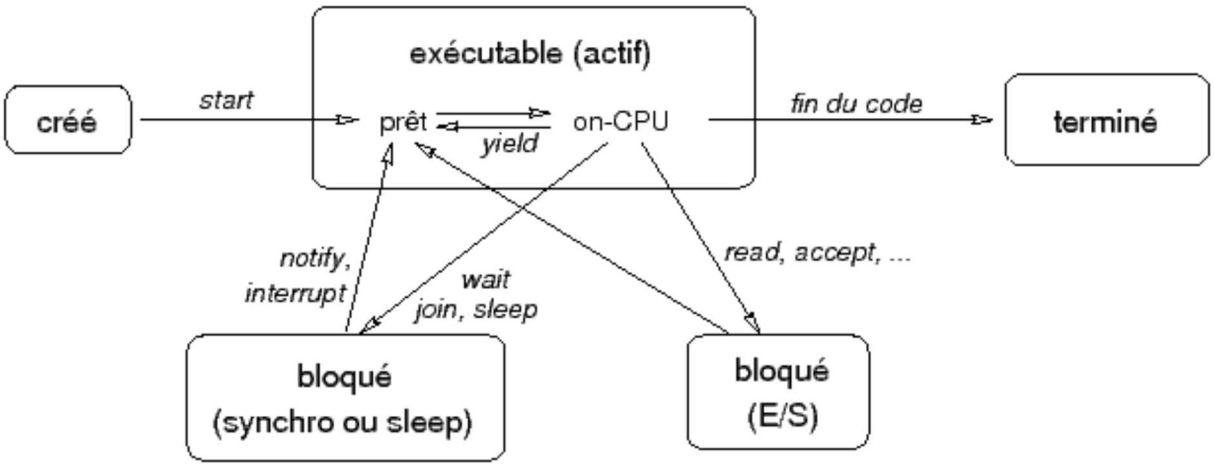
Les threads en Java

Ce document est aussi disponible en postscript.

1 Généralités

La machine Java fournit le support d'un noyau de gestion d'activités (ou processus légers, ou threads). Lorsqu'une machine Java est démarrée, elle crée une première activité applicative (qui appelle la procédure `main` de la classe spécifiée), puis des activités peuvent être créées et détruites dynamiquement. La méthode `System.exit` permet d'arrêter globalement la machine par un appel explicite. Sinon, l'exécution de la machine se poursuivra jusqu'à la terminaison de toutes les activités applicatives. Hormis ces activités applicatives, un certain nombre de activités dites *démoniques* gèrent des activités de supervision (ramasse-miettes par exemple).

Le cycle de vie d'une activité Java est similaire au cycle de vie standard d'une activité des Threads Posix, ou d'un processus :



Une activité est caractérisée par les attributs suivants :

- un nom externe (`String`), défini à la création (explicitement ou avec une valeur par défaut), accessible par `Thread.getName()` et modifiable avec `Thread.setName(String)`.
- état démonique : hérité de l'activité créatrice, est testé avec `Thread.isDaemon()`, et positionné *avant le démarrage de l'activité* par `Thread.setDaemon(bool)`.
- priorité : héritée de l'activité créatrice, est consultée et modifiée par `Thread.getPriority()` et `Thread.setPriority(int)`. Cette priorité est utilisée pour l'ordonnancement à court terme (partage du processeur entre les activités exécutables) : quand la machine Java est disponible et doit sélectionner une nouvelle activité, elle choisit une (au hasard) des activités ayant la priorité la plus élevée. A priori, la machine Java n'est pas préemptible (cf 4.3).
- appartenance à un groupe : défini à la création (explicitement ou par héritage de l'activité créatrice), accessible par `Thread.getThreadGroup()` (non modifiable). Voir 5.2.

2 Définition et création d'une activité

Une activité peut être créée de deux manières :

- Héritage de la classe `Thread` : on définit une classe qui hérite de `Thread` et (re)définit la méthode `run` :

```
class X extends Thread {
   ⋮
   public void run () {
       … code de l'activité …
   }
}

// Utilisation
foo() {
   X x = new X();
   x.start();
   ⋮
   x.join();
}
```

- Implantation de l'interface `Runnable` : on définit une classe qui implante l'interface `Runnable` (ce qui consiste simplement à définir une procédure `public void run()`), et on crée une instance de `Thread` avec un objet `Runnable` :

```
class X implements Runnable {
   ⋮
   public void run () {
       … code de l'activité …
   }
}

// Utilisation
foo() {
   X x = new X(…);
   Thread t = new Thread(x);
   t.start();
   ⋮
   t.join();
}
```

- Pièges :
  - distinguer la méthode `run` (qui est le code exécuté par l'activité) et la méthode `start` (méthode de la classe `Thread` qui rend l'activité exécutable) ;
  - dans la première méthode de création, attention à définir la méthode `run` avec strictement le prototype indiqué (il faut redéfinir `Thread.run` et non pas la surcharger).

3 Opérations de la classe `Thread`

3.1 Les constructeurs

Une activité est créée en fournissant plus ou moins de paramètres explicites. Trois éléments interviennent : le groupe, l'objet de la classe `Runnable` fournissant le code à exécuter, un nom externe.

Par défaut, un nom externe de la forme `"Thread-"+<entier>` est attribué à l'activité.

```
Thread(ThreadGroup, Runnable, String);
Thread(ThreadGroup, Runnable);
Thread(ThreadGroup, String);
Thread(Runnable, String);
Thread(Runnable);
Thread(String);
Thread();
```

3.2 Méthodes de classe

- `static Thread currentThread()` permet d'obtenir l'activité appelante ;
- `static void yield()` laisse une chance aux autres activités de s'exécuter ;
- `static void sleep(long millisec) throws InterruptedException` suspend l'exécution de l'activité appelante pendant la durée indiquée ou jusqu'à ce que l'activité soit interrompue (voir 3.4) ;

3.3 Vie de l'activité

- `void start()` rend l'activité exécutable après sa création ;
- `void join() throws InterruptedException` attend que l'activité soit terminée ;
- `void join(long millisec) throws InterruptedException` attend que l'activité soit terminée ou que le délai de garde soit écoulé.

3.4 Interruption

La classe `Thread` fournit un mécanisme minimal permettant d'interrompre une activité : la méthode `interrupt` (appliquée à une activité) provoque la levée de l'exception `InterruptedException` si l'activité est bloquée sur une opération de synchronisation (suite à un appel à `Object.wait`, `Thread.join` ou `Thread.sleep`). Sinon, un indicateur `interrupted` est positionné. Cet indicateur est testé par deux méthodes :

- `boolean isInterrupted()` qui renvoie la valeur de l'indicateur de l'activité sur laquelle cette méthode est appliquée ;
- `static boolean interrupted()` qui renvoie et *efface* la valeur de l'indicateur de l'activité appelante.

Noter que ce mécanisme ne permet pas d'interrompre une entrée-sortie bloquante en cours (comme une lecture en attente de donnée) : l'indicateur d'interruption est positionné, mais aucune exception n'est levée et l'activité reste bloquée. Ce point limite considérablement l'intérêt de ce mécanisme.

4 Problèmes et difficultés

4.1 Activités + Objets ≠ Acteurs

En dépit du mode de création, l'activité n'est pas associée à l'objet qui a servi à la créer. Considérons l'exemple suivant :

```
class X extends Thread {
  public void foo() { System.out.println(Thread.currentThread().getName()); }
  public void run() {
        this.foo();       // (1)
  }
}
class Y extends Thread {
  X unX;
  Y (X _x) { unX = _x; }
  public void run() {
         unX.foo();      // (2)
  }
}
public class ConfusionActeurs {
  public static void main (String[] unused) {
      X x = new X();
      x.setName("T1");
      x.start();
```

```
      Y y = new Y(x);
      y.setName ("T2");
      y.start();
      x.foo();          // (3)
  }
}
```

L'appel 1 produit `T1`, l'appel 2 produit `T2` et l'appel 3 produit `main`, alors que tous s'appliquent au même objet. C'est pourquoi, il est souvent préférable d'utiliser la deuxième forme de création (implantation de `Runnable`), qui évite la confusion entre l'activité et l'objet.

4.2 Absence de suicide

La classe `Thread` ne prévoit qu'une seule cause de terminaison d'une activité : quand l'exécution du code associé (méthode `run`) est terminé (que ce soit en atteignant proprement la fin de la procédure, par un `return` placé dans `run`, ou à cause d'une exception non capturée, levée dans une méthode appelée depuis `run`, ce dernier cas entraînant l'arrêt de la machine virtuelle). Il est cependant possible de réaliser le suicide ainsi :

```
class ThreadSuicide extends Error {
  // Error est une forme de Throwable qu'il n'est pas nécessaire de
  // déclarer dans la clause throws des méthodes qui la lèvent.
  // Sauf cas très particulier (comme ici), elle ne doit jamais être capturée.
   public static void exit() {
     throw new ThreadSuicide();
   }
}

class X extends Thread {
   public void run () {
     try {
       … foo(); …
     } catch (ThreadSuicide e) {
     }
}

void foo() {  // dans n'importe quelle classe
  ⋮
  if (! bon)
     ThreadSuicide.exit();
}
```

4.3 Préemption et ordonnancement à court terme

La spécification de Java est très imprécise sur la préemption et l'ordonnancement des activités. La seule chose qu'exige la norme est la gestion des priorités telle que décrite en 1 : quand la machine Java est disponible et doit sélectionner une nouvelle activité, elle choisit une (au hasard) des activités ayant la priorité la plus élevée. La machine Java devient disponible quand une activité se bloque (appel à `Object.wait`, `Thread.join` ou `Thread.sleep`) ou accepte de céder le processeur (appel à `Thread.yield`).

Deux points sont donc problématiques : que se passe-t-il quand une activité se bloque sur une entrée-sortie ? Que se passe-t-il si une activité de calcul faiblement prioritaire ne relâche pas volontairement le processeur ?

En pratique, il existe (au moins !) deux implantations des Threads dans java :

- les « green threads », implantés en interne à la JVM (machine virtuelle Java), non préemptif, mais assurant la commutation sur entrée-sortie bloquante ;
- les « native threads », utilisant une bibliothèque dédiée au matériel sur lequel s'exécute la JVM. Dans le cas de java 1.2 sur Solaris, la bibliothèque utilisée est celle des Threads Posix, ce qui assure le

non-blocage de la JVM sur E/S, la préemption avec partage de l'accès au processeurs, et l'utilisation éventuelle de plusieurs processeurs matériels. Ce mode est le mode par défaut.

5 Divers

5.1 Variables localisées

Outre les références globales (attributs des objets, visibles par toutes les activités) et les variables locales (visibles uniquement au sein de la fonction et par l'activité appelante), il existe des variables globales ayant une valeur distincte dans chaque activité. Le *nom d'une activité* peut être perçu comme une variable localisée[1].

De telles variables sont des instances de `ThreadLocal` ou de `InheritableThreadLocal` qui fournissent l'interface suivante :

- `public Object get()` permet d'obtenir la valeur de la variable localisée pour l'activité appelante ;
- `public void set(Object)` permet de positionner la valeur de la variable localisée pour l'activité appelante ;
- `protected Object initialValue()` peut être redéfinie pour qu'une activité dispose d'une valeur par défaut autre que `null`.

Remarquer qu'il n'est pas possible de consulter ou de modifier la valeur d'une variable localisée d'une autre activité.

L'exemple suivant crée des activités qui disposent chacune d'un numéro différent (déterminé à leur premier appel à `numero.get()`). Le numéro de l'activité courante est obtenu depuis n'importe quelle méthode en utilisant `DemoThreadLocal.numero.get()`.

```
class NumeroThread extends ThreadLocal {
  static int numeroCourant = 0;
  protected Object initialValue() {
     numeroCourant++;
     return new Integer(numeroCourant);
  }
}
class Activite implements Runnable {
  public void run() {
     System.out.println("Mon numero est " + (Integer) DemoThreadLocal.numero.get());
  }
}
public class DemoThreadLocal {
  static NumeroThread numero = new NumeroThread();
  public static void main(String[] unused) {
     for (int i = 0; i < 5; i++) {
        Activite a = new Activite();
        new Thread(a).start();
     }
  }
}
```

5.2 Groupes d'activités

Les activités peuvent être structurées en groupes (classe `ThreadGroup`). Implicitement, toutes les activités appartiennent au groupe système (racine). D'autres groupes peuvent être créés. Une hiérarchie peut exister entre les groupes selon une structure d'arbre. Cette notion permet en particulier de déclencher une opération sur tous les membres d'un groupe (changement de la priorité des activités du groupe, énumération des activités du groupes…).

5.3 Exercice

Écrire un programme qui contient trois activités :

- l'activité A affiche "toto" toutes les deux secondes ;
- l'actitité B additionne itérativement les nombres de 1 à 5000 ;
- l'actitité C affiche "truc" toutes les trois secondes ;
- le programme principal attend la terminaison de B et termine alors l'application.

Proposer des solutions avec et sans `join`.

6 La synchronisation

Les activités interagissent lorsqu'elles entrent en concurrence pour l'accès à des objets communs ou quand elles coopèrent via des objets partagés. De façon classique, on trouve deux niveaux de synchronisation : d'une part, le problème de l'exclusion mutuelle d'accès à des données (objets) ou à du code (des méthodes), d'autre part, le problème de la synchronisation sur des événements. La combinaison des deux forme dans Java des moniteurs de Hoare dégénérés.

6.1 L'exclusion mutuelle

Pour traiter les problèmes d'exclusion mutuelle, Java propose la définition de sections critiques exprimées à l'aide du mot clé `synchronized`.

- Tout objet Java est équipé d'un verrou d'exclusion mutuelle. Ainsi, pour assurer qu'une seule activité accède à un objet `unObj` d'une classe quelconque, on définit les actions sur l'objet dans une région critique par la syntaxe :

```
synchronized (unObj) {
    < Région critique >
}
```

- Une méthode peut aussi être qualifiée de `synchronized` :

```
synchronized T uneMethode(…) { … }
```

  Ceci est équivalent à :

```
T uneMethode(…) {
     synchronized (this) { … }
}
```

  Il y a donc exclusion d'accès de l'objet sur lequel on applique la méthode, pas de la méthode elle-même, qui peut être exécutée concurremment sur des objets différents.

- Chaque classe possède aussi un verrou exclusif qui s'applique aux méthodes de classe (méthodes statiques) :

```
class X {
   static synchronized T foo() { … }
   static synchronized T' bar() { … }
}
```

  L'utilisation de `synchronized` assure ici l'exécution en exclusion mutuelle pour toutes les méthodes *statiques* synchronisées de la classe `X`. Noter que ce verrou ne concerne pas l'exécution des méthodes d'objets.

- Les verrous sont qualifiés de *récursifs* ou *réentrants* : si une activité possède un verrou, une deuxième demande provenant de cette activité est satisfaite sans causer d'auto-interblocage, et le code suivant s'exécute sans problème :

```
synchronized(o) { … synchronized(o) { … } … }
```

En programmation « propre », on évitera cependant de dépendre de cela.

6.2 L'interblocage dû aux verrous

Avec l'utilisation de plusieurs verrous, le risque d'interblocage existe, dès qu'une activité peut posséder plusieurs verrous. Par exemple, le code suivant ne garantit pas l'absence d'interblocage :

```
synchronized (o1) { synchronized (o2) { … } }
           ||
synchronized (o2) { synchronized (o1) { … } }
```

La solution pour garantir l'absence d'interblocage par les verrous est la « stratégie par classes ordonnées » : on définit une relation d'ordre (d'importance) sur tous les verrous et on assure que les activités acquièrent les verrous exclusivement par ordre croissant d'importance. Dans l'exemple précédent, si `o1` est moins important que `o2`, la deuxième ligne est erronée. En général, il est aisé de vérifier qu'un code proprement écrit respecte la contrainte d'ordre.

6.3 La synchronisation par objet

Pour synchroniser des activités sur des conditions logiques, on dispose du couple d'opérations permettant d'assurer le blocage et le déblocage des activités, en l'occurrence (`wait`, `notify[All]`). Ces opérations sont applicables à tout objet, pour lequel l'activité a obtenu au préalable l'accès exclusif. L'objet est alors utilisé comme une sorte de variable condition.

- `unObj.wait()` libère l'accès exclusif à l'objet et bloque l'activité appelante en attente d'un réveil via une opération `unObj.notify` ;
- `unObj.notify()` réveille une seule activité bloquée sur l'objet (si aucune activité n'est bloquée, l'appel ne fait rien) ;
- `unObj.notifyAll()` réveille toutes les activités bloquées sur l'objet.

L'opération `wait` peut aussi se terminer par une interruption (3.4) ou après un délai de garde spécifié à l'appel.

Dans tous les cas, lorsqu'une activité est réveillée, elle est mise en attente de l'obtention de l'accès exclusif à l'objet.

6.4 Implantation des sémaphores

```
public class Semaphore {

    private int cpt = 0;
    Semaphore (int c) {
        cpt = c;
    }

    public void P() throws InterruptedException {
        synchronized (this) {
            while (cpt == 0) {
                this.wait ();
            }
            cpt--;
        }
    }

    public void V() {
        synchronized (this) {
            cpt++;
            this.notify ();
```

```
        }
    }
}
```

6.5 Difficultés

- Attention aux prises multiples de verrous :

  ```
  synchronized (o1) { synchronized (o2) { o1.wait(); } }
  ```

  Dans ce cas, l'appel `o1.wait` ne libère que le verrou exclusif de `o1`, alors que le verrou exclusif de `o2` reste acquis à l'activité bloquée.
- L'existence d'une seule notification possible pour une exclusion mutuelle donnée rend difficile la résolution de problèmes de synchronisation. En dépit d'une apparence similaire, les solutions inspirées des moniteurs de Hoare sont difficilement transposables si elles nécessitent plus d'une variable condition. Deux voies sécurisés s'offrent au pauvre programmeur Java : soit utiliser les sémaphores dont l'implantation a été fournie ci-dessus ; soit affecter un objet de blocage distinct à chaque requête et gérer soit-même les files d'attente.
- Il n'existe aucun ordonnancement garanti pour l'ordre de réveil des activités bloquées. En outre l'activité réveillée ne devient effectivement active que lorsqu'elle a reobtenu le verrou d'exclusion mutuelle, et l'état peut avoir été modifié entre temps. Il est donc indispensable de retester la condition de blocage (boucle `while` dans l'exemple ci-dessus).
- Compte tenu des remarques précédentes, et en ajoutant le peu de garantie sur le partage processeur (4.3), garantir la vivacité des activités est difficilement soluble.

6.6 Gestion explicite des files d'attente

La solution la plus simple (mais pas la plus élégante) pour résoudre réellement un problème de synchronisation en Java consiste en la gestion explicite des requêtes bloquées. On définit ainsi une classe Requête, qui contient les paramètres de demande. Quand une requête ne peut pas être satisfaite, on crée un nouvel objet Requête, on le range dans une structure de données, et l'activité demandeuse se bloque sur l'objet Requête. Quand une activité modifie l'état de sorte qu'il est possible qu'une (ou plusieurs) requête soit satisfaite, elle parcourt les requêtes en attente pour débloquer celles qui peuvent l'être. La condition de satisfaction et la technique de parcours permet d'implanter précisément la stratégie souhaitée.

La première difficulté provient de la protection des variables partagées par toutes les activités (état du système et des files d'attente) tout en assurant un blocage indépendant ; cela conduit à l'apparition d'une « fenêtre », où une activité tente de débloquer une autre activité avant que celle-ci n'ait effectivement pu se bloquer. La deuxième difficulté réside dans l'absence d'ordonnancement lors des réveils, ce qui nécessite que la mise-à-jour de l'état soit faite dans l'activité qui réveille et non pas dans l'activité qui demande. On obtient alors la structure suivante (en italique, ce qui concerne spécifiquement le problème résolu : l'allocateur de ressources) :

```
class Allocateur {

    private class Requête {
        boolean estSatisfaite = false;
        int nbDemandé;          // paramètre d'une requête
        Requête (int nb) { nbDemandé = nb; }
    }

    // les requêtes en attente de satisfaction
    java.util.List lesRequêtes = new java.util.LinkedList();
    int nbDispo = …;        // le nombre de ressources disponibles

    void allouer (int nbDemandé) throws InterruptedException
    {
        Requête r = null;
        synchronized (this) {
```

```
            if (nbDemandé <= this.nbDispo) { // la requête est satisfaite immédiatement
                this.nbDispo -= nbDemandé;    // maj de l'état
            } else {            // la requête ne peut pas être satisfaite
                r = new Requête (nbDemandé);
                this.lesRequêtes.add (r);
            }
        }
        // fenêtre => nécessité de estSatisfaite (plus en excl. mutuelle donc une autre
        // activité a pu faire libérer, trouver cette requête et la satisfaire avant
        // qu'elle n'ait eu le  temps de se bloquer effectivement).
        if (r != null) {
            synchronized (r) {
                if (! r.estSatisfaite)
                  r.wait();
                // la mise à jour de l'état se fait dans le signaleur.
            }
        }
    } // allouer

    public void libérer (int nbLibéré)
    {
        synchronized (this) {
            this.nbDispo += nbLibere;
            // stratégie bourrin : on réveille tout ce qu'on peut.
            java.util.Iterator it = lesRequêtes.iterator();
            while (it.hasNext()) {
                Requête r = (Requête) it.next();
                synchronized (r) {
                    if (r.nbDemandé <= this.nbDispo) { // requête satisfaite !
                        it.remove();
                        this.nbDispo -= r.nbDemandé; // maj de l'état
                        r.estSatisfaite = true;
                        r.notify();
                    }
                }
            }
        }
    } // libérer
}
```

---

[1] La norme Posix Threads utilise le terme de *specific data*.

Ce document a été traduit de LaTeX par HeVeA